Pasi Härme

REST-rajapintalähtöisen ohjelmisto-arkkitehtuurin suunnittelu PlanMill-toiminnanohjausjärjestelmään

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

8.4.2015

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Pasi Härme REST-rajapintalähtöisen ohjelmistoarkkitehtuurin suunnittelu PlanMill-toiminnanohjausjärjestelmään 30 sivua 8.4.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Senior Consultant/Manager Marjukka Niinioja Lehtori Simo Silander

Tämän insinöörityön tavoitteena oli tutustua REST-arkkitehtuurimalliin ja suunnitella ja toteuttaa opitun pohjalta uusi REST-rajapintalähtöinen sovellusarkkitehtuuri PlanMill-toiminnanohjausjärjestelmälle. Työn tilaajana toimi PlanMill Oy.

Jotta työn mittakaava olisi säilynyt järkevänä, työssä ei toteutettu koko nykyisen järjestelmän mittaista sovellusta, vaan se rajoitettiin kattamaan yhden järjestelmän käytetyimmistä toiminnallisuuksista: tuntiraporttien kirjauksen. Työn ulkopuolelle jätettiin myös autentikaatio ja nykyiselle PlanMill-järjestelmälle oleellinen parametrijärjestelmä.

Koska nykyinen PlanMill-järjestelmä on toteutettu käyttäen Java-teknologioita ja itsellänikin on ohjelmointikielistä siitä eniten kokemusta, oli Javan käyttö uuden arkkitehtuurin pohjana luonnollinen valinta. Sovelsin muutamia hyviksi todettuja vapaan lähdekoodin kirjastoja arkkitehtuurin rungon rakentamiseksi.

Lopputuloksena syntyi sovellusarkkitehtuuri, joka tarjoaa REST-mallin mukaisen web-rajapinnan tuntiraporttien kirjaamiseen, muokkaamiseen ja poistamiseen.

Avainsanat	web-rajapinta, rest, http, ohjelmistoarkkitehtuuri

Author(s) Title Number of Pages Date	Pasi Härme Designing a REST Interface Oriented Software Architecture for PlanMill PSA 30 pages 8 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Marjukka Niinioja, Senior Consultant/Manager Simo Silander, Senior Lecturer

The original goal of this thesis was to research and implement new RESTful web API-driven software architecture for the PlanMill PSA. The project was ordered by PlanMill Ltd.

Because we wanted to keep the scale of the project reasonable, it was limited to only cover one of the most used features of the current PlanMill system: time reporting. Authentication and the current PlanMill-system's essential parameter handling were also left out of scope.

Because the current system is built on Java technology, which I myself have some experience of, the use a Java as the base of the new architecture was a natural choice. I applied some of the most well-proven open source libraries to form the backbone of the new architecture.

The end result was a software architecture which provides a RESTful web service for creating, modifying and deleting timereports.

Keywords	web-api, web service, rest, http, program architecture

Sisällys

Lyhenteet

ASP	Active Server Pages. Microsoftin kehittämä web-ohjelmointikehys dynaamisten web-palvelujen rakentamiseen.
CRUD	Create, Read, Update, Delete. Pysyvän tiedon tallentamisen, lukemisen, päivittämisen ja poistamisen mahdollistavat operaatiot.
HTTP	Internetsisällön siirrossa yleisimmin käytetty protokolla, joka toimii perinteisellä asiakas-palvelin-kommunikointiperiaatteella.
JPA	Java Persistence API. Java-ohjelmointirajapinta relaatiomallisen tiedon hallintaan.
JPQL	Java Persistence Query Language. JPA-spesifikaation määrittämä oliopohjainen tietokantakyselykieli.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto tiedonvälitykseen. Nimestään huolimatta se on täysin riippumaton JavaScriptista.
JSP	Java Server Pages. Sun Microsystemsin kehittämä vaihtoehto ASP:lle ja PHP:lle, jolla voidaan luoda HTML- tai XML-muotoisia websivuja.
JVM	Java Virtual Machine. Virtuaalikone, jossa voidaan ajaa Javan tavukoodiksi käännettyjä ohjelmia.
MVC	Model-View-Controller (malli-näkymä-käsittelijä). Ohjelmistoarkkitehtuuri jossa käyttöliittymä erotetaan sovellusaluetiedosta.
ORM	Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.
PHP	Personal Home Page. Palvelinpuolen skriptikieli web-palvelujen kehitykseen.

PSA	Professional Services Automation. Asiantuntijayritysten toiminnanohjausjärjestelmä. Sitä voidaan kutsua palveluyritysten toiminnanohjausjärjestelmäksi.
REST	Representational State Transfer. HTTP-protokollaan perustuva arkkitehtuurimalli web-palvelujen rajapintojen toteuttamiseksi.
SaaS	Software as a Service. Ohjelmisto, jota tarjotaan palveluna perinteisen lisenssiostotavan sijaan. Asiakkaat käyttävät SaaS-palvelua yleensä verkkoselaimessa ajettavan asiakasohjelman kautta.
SOAP	SOAP on Microsoftin kehittämä tietoliikenneprotokolla web-palveluille, jotka mahdollistavat proseduurien etäkutsumisen.
URI	Uniform Recource Identifier. Merkkojono, joka kertoo jonkin resurssin paikan tai sen yksikäsitteisen nimen. URI:n erikoistapausta URL:ia käytetään web-resurssien osoittamiseen.
URL	Uniform Resource Locator. Merkkijono, jolla kerrotaan jonkin resurssin sijainti. Käytetään erityisesti osoittamaan internetin WWW-sivuja.
WSDL	Web Services Description Language. XML-perustainen kieli web-palvelujen kuvaamiseen.
XML	Extensive Markup Language. Merkintäkielien standardi, jossa metatietoa voidaan sisällyttää tiedon sekaan.

1 Johdanto

Alun perin tämän työn tarkoituksena oli suunnitella ja pilotoida PlanMill Oy:n toiminnanohjausjärjestelmälle uusi REST-arkkitehtuurimallia noudattava web-rajapinta PlanMill-sovelluksen käyttämiseksi. Insinöörityön tavoitteena oli tutkia tapoja ja tekniikoita, miten nykyistä rajapintaa voidaan parantaa asiakaspalautteen pohjalta ja todeta, mikä niistä on sopivin.

Rajapinnan suunnitteluvaiheessa kuitenkin todettiin, että uuden REST-rajapinnan toteuttaminen nykyisen PlanMill-järjestelmän päälle tulisi olemaan hyvin vaikeaa. Tämän johdosta lopputyön alkuperäistä aihetta päätettiin laajentaa pidemmälle. Pelkän web-rajapinnan sijaan projekti kattaa koko palvelinarkkitehtuurin suunnittelun ja toteutuksen. Tarkoituksena on siis toteuttaa proof of concept -mallinen pilotti PlanMill-sovelluksen uudesta backend-ohjelmistoarkkitehtuurista. Insinöörityön teorian painopiste pysyy kuitenkin aidosti REST-mallia noudattavan web-rajapinnan suunnittelussa toteuttamisessa.

Ensiksi tässä työssä tutustutaan REST-arkkitehtuurimallin teoriaan ja sen soveltamiseen web-rajapinnan toteuttamisessa HTTP-protokollaa hyväksi käyttäen, jonka jälkeen käydään läpi PlanMill-järjestelmän tarjoama Open API -rajapinta ja sen sisältämät nykyiset epäkohdat.

Seuraavaksi siirrytään uuden arkkitehtuurin tarpeisiin ja sen kautta käydään läpi arkkitehtuurin toteutuksessa käytetyt teknologiat. Lopuksi käydään läpi uusi arkkitehtuuri pääkomponentteineen ja niiden toiminnallisuudet kokonaisarkkitehtuurin kannalta.

PlanMill on suomalainen pk-yritys, joka erikoistuu verkkopohjaisen toiminnanohjausovelluksen toimittamiseen. PlanMill-järjestelmää tarjotaan pääsääntöisesti SaaS-mallin (Software as Service) mukaisesti vuokrattavana verkkosovelluksena, mutta asiakkaan omalle palvelimelle toimitettava on-premise-asennus on myös mahdollista.

PlanMilliä käyttää yli sata asiakasyritystä, joista koostuu yli 20000 käyttäjää 25 eri maassa. Asiakasyritykset ovat suureksi osaksi asiantuntijapalveluja tarjoavia yrityksiä,

jotka toimivat esimerkiksi taloushallinnon, sovelluskehityksen, lakitieteen tai mediatuotannon alalla.

PlanMill Oy perustettiin vuonna 2001 TJ Groupin tytäryhtiöksi. Yrityksestä tuli itsenäinen vuonna 2006 ja sen nykyisiä omistajia ovat Thomas Hood, Christer Nordlund sekä sijoitusyhtiö Canelco Capital Oy.

2 REST-arkkitehtuurimalli ja sen soveltaminen internetympäristössä

REST on termi, jolla Roy Fieldinging [1, s. 76] kutsuu väitöskirjassaan verkkopohjaisten järjestelmien arkkitehtuuria. REST tulee sanoista “Representational State Transfer”. REST ei itsessään ole arkkitehtuuri vaan malli tai ohjenuora, jota noudattamalla voidaan toteuttaa RESTful-tyyppinen web-rajapinta. Se ei siis ota kantaa itse rajapinnan toteutukseen vaan keskittyy tarjoamaan joukon rajoitteita ja määrityksiä rajapinnan toteuttamiseksi. Internet edustaa suurinta toteutusta järjestelmästä, joka noudattaa REST-arkkitehtuurimallia.

REST-malli on lähtökohtaisesti suunniteltu skaalautuvien web-palvelujen toteuttamiseen. Se koostuu joukosta rajoitteita, jotka mahdollistavat suorituskykyisen ja helposti ylläpidettävän arkkitehtuurin luomisen. REST-palvelut kommunikoivat yleensä HTTP-protokollan välityksellä käyttäen hyväksi sen tarjoamia metodeja (kuten GET, POST, PUT, DELETE ja PATCH) yksinkertaisten CRUD-operaatioiden toteuttamikseksi tiedon manipulointia varten.

RESTiä pidetään yksinkertaisempana vaihtoehtona SOAP- ja WSDL-tyyppisille web-palveluille. Ennen REST-mallia SOAP oli yleisesti käytetty protokolla web-palvelujen tiedonsiirrossa palvelimen ja asiakkaiden välillä. SOAP käyttää tiedonsiirtoon monimutkaisia XML-dokumentteja viestimiseen. Näiden dokumenttien rakentaminen voi olla hankalaa, koska joitain ohjelmointikieliä käytettäessä ne joudutaan luomaan käsin ja sen lisäksi SOAPilla on myös hyvin alhainen virheensietokyky.

Web-rajapintojen näkökulmasta REST on helpommin lähestyttävä arkkitehtuuri kuin SOAP. RESTin ei tarvitse niin sanotusti keksiä pyörää uudelleen, koska HTTP-protokolla sisältää itsessään suuren osan työkaluista REST-arkkitehtuurimallisen raja-

pinnan toteuttamiseen. REST ei ole myöskään riippuvainen mistään teknologioista ja se on hyvin skaalautuva, yksinkertainen ja helposti laajennettava.

2.1 REST

Roy Fieldingin mukaan arkkitehtuurisuunnitteluun on olemassa kaksi lähestymistapaa: Ensimmäisessä aloitetaan täysin tyhjältä pöydältä. Arkkitehtuuria rakennetaan lisäämällä siihen pieniä komponentteja, kunnes se tyydyttää arkkitehtuurille annetut ehdot. Toisessa tavassa lähtökohtana ovat suunniteltavan järjestelmän vaatimukset kokonaisuudessaan ilman mitään rajoitteita, eli niin sanottu ”Null style”. Arkkitehtuurisuunnittelija tunnistaa ja soveltaa rajoitteita järjestelmän eri osiin. Ensimmäinen lähestymistapa painottaa luovuutta ja rajoittamatonta visiota, kun jälkimmäinen lähestymistapa korostaa rajoitteita ja järjestelmän kontekstin ymmärtämistä. REST-arkkitehtuurimalli on suunniteltu käyttäen jälkimmäistä lähestymistapaa. Jos järjestelmä täyttää kaikki Fieldingin arkkitehtuurille asettamat rajoitteet, voidaan sitä kutsua REST-arkkitehtuurin mukaiseksi. [1, s. 77.] Seuraavaksi käydään läpi nämä rajoitteet.

2.1.1 Asiakas-palvelin-malli

Asiakas-palvelin-mallin periaatteena on käyttöliittymän ja tiedonkäsittely- ja tallennuslogiikan eriyttäminen. Tämä tekee käyttöliittymästä paremmin siirrettävän eri alustoille ja parantaa palvelinkomponenttien skaalautuvuutta. Asiakas- ja palvelinohjelmistoja voidaan myös kehittää toisistaan riippumatta, kunhan niiden välinen rajapinta ei muutu. [1, s. 78.]

2.1.2 Tilattomuus

Edellisen rajoitteen päälle asetetaan rajoite, joka määrää, että kaikki kommunikointi asiakkaan ja palvelimen välillä pitää olla tilatonta. Tämä tarkoittaa käytännössä sitä, että jokaisen asiakkaan lähettämän pyynnön pitää sisältää kaikki tarpeellinen informaatio pyynnön käsittelyä varten, koska palvelimella ei säilytetä asiakkaan istuntoa. Istunnon tilan hallinta jää täten kokonaan asiakkaan hoidettavaksi. [1, s. 78-79.]

2.1.3 Välimuistin käyttö

Koska jokaisen asiakkaan ja palvelimen välisen interaktioiden täytyy olla täysin tilattomia, jokaisen asiakkaan lähettämän kutsun pitää sisältää kontekstiin liittyvää informaatiota, joka voi suurella mittakaavalla kuormittaa palvelinta.

Tilattomuus mahdollistaa kuitenkin tavan rajoittaa palvelimen kuormittamista käyttämällä hyväksi välimuistia, joka voi sijaita asiakasohjelmassa tai vaikka välityspalvelimella. Sen sijaan että asiakas pyytäisi jo aikaisemmin haetun resurssin palvelimelta, voi se hakea resurssin välimuistista, jos palvelin on sallinut sen sinne tallettamisen. Huonona puolena välimuistissa on luotettavuus, koska ei ole varmaa, onko välimuistissa sijaitseva resurssi vanhentunut vai ei. [1, s. 79-81.]

2.1.4 Yhdenmukainen rajapinta

Oleellisin piirre REST-arkkitehtuurimallissa, joka erottaa sen muista web-arkkitehtuureista, on sen panostus yhdenmukaiseen rajapintaan järjestelmän eri komponenttien välillä. Kun sovelletaan ohjelmistokehityksen geneerisyys-periaatetta rajapinnalle, koko järjestelmäarkkitehtuuri pysyy yksinkertaisena, ja komponenttien välisen kanssakäymisen näkyvyys paranee. Huonona puolena yhdenmukaisessa rajapinnassa on sen käsittelemän tiedon standardisoitu muoto, joka ei ole välttämättä täysin optimaalinen rajapintaa käyttäville sovelluksille.

REST-malli tarkentaa yhdenmukaisen rajapinnan käsitettä kolmella lisärajoitteella, jotka ovat resurssien tunnistaminen, resurssien manipuloiminen esitysten avulla sekä itseselitteiset viestit.

REST-arkkitehtuurimallissa resurssilla tarkoitetaan jotain nimettävää kohdetta tai käsitettä, esimerkiksi dokumenttia, kuvaa, resurssikokoelmaa tai konkreettista objektia kuten vaikka henkilötietoja. Resurssin esitys ei välttämättä aina pysy samana vaan se voi muuttua olosuhteiden mukaan. Esimerkiksi voidaan kuvitella, että ”viimeisin tuntikirjaus” on resurssi, jonka esitys todennäköisesti muuttuu suhteellisen usein. Myös ”tuntikirjaus X” on staattinen resurssi. Edellä mainitut esimerkit ovat eri resursseja, vaikka onkin mahdollista, että ne molemmat sisältävät saman esityksen. Jokaiseen resurssiin kuuluu yksiselitteinen tunniste, jonka avulla resurssin voi löytää.

Resurssiin kuuluu vähintäänkin yksi esitys. Se voi olla esimerkiksi JSON- tai XML-dokumentti tai binääritiedosto kuten kuva. Jotkin internetin REST-rajapinnat antavat mahdollisuuden tuottaa ja vastaanottaa resurssien esityksiä useammassa kuin yhdessä formaatissa. Resurssin esitykseen liittyy myös metadataa, joka kuvaa esityksen standardin ja määrittelee sen rakenteen ja semantiikan. Kun asiakasohjelmalla on hallussaan koko resurssin esitys ja siihen liittyvä metatieto, on sillä tarpeeksi informaatiota sen käsittelemiseen.

Viestien itseselitteisyydellä tarkoitetaan sitä, että kaikki viestin käsittelyyn tarvittava tieto liitetään itse viestiin. Tämä mahdollistaa välikomponenttien käytön viestien prosessoinnissa. Viestien itseselitteisyys on seuraus aiemmin mainitusta tilattomuusrajoitteesta, joka edellyttää kaiken pyynnön käsittelyyn tarvittavan tiedon sisällyttämistä sen sisään. [1, s. 81-82.]

2.1.5 Kerroksittainen järjestelmä

Kerroksittainen järjestelmä mahdollistaa arkkitehtuurin muodostamisen hierarkkisista tasoista niin, että kunkin tason komponentit ovat tietoisia vain suoraan sen ylä- tai alatasolla sijaitsevista komponenteista. Näin voidaan esimerkiksi kapseloida ns. legacy-järjestelmiä rajapintojen taakse tai suojella uusia palveluita vanhentuneilta asiakasohjelmilta. Toinen esimerkki on käyttää yhtenä tasona kuormantasaajaa, joka tasaa verkkoliikennettä alemmille tasoille.

Suurin haittapuoli monitasoisessa järjestelmässä on viive, joka ilmenee, kun tietoa prosessoidaan monella tasolla. Viivettä voidaan kuitenkin rajoittaa käyttämällä hyväksi jaettua välimuistia eri kerroksien välillä. [1, s. 82-84.]

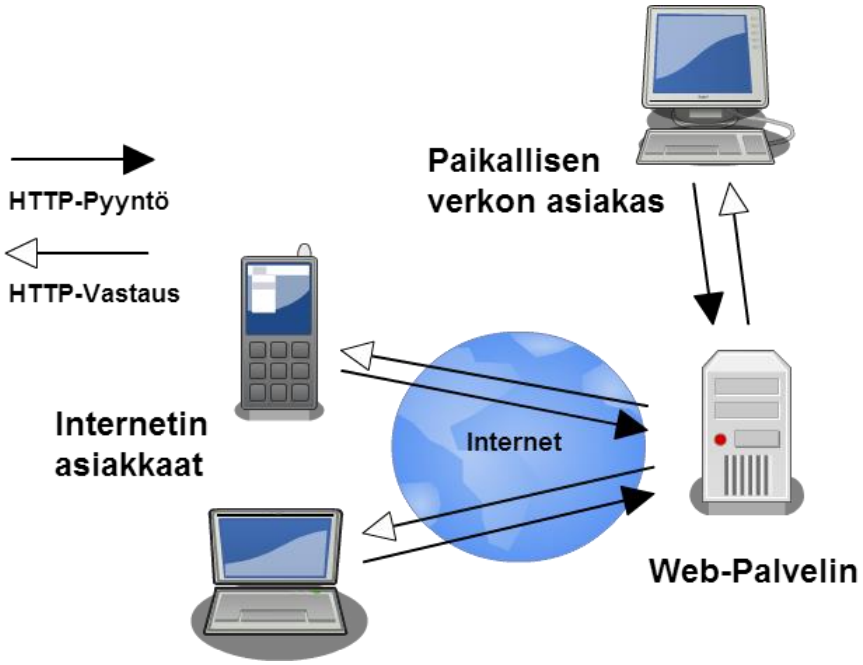
2.1.6 Code on demand

Code on Demand eli ”Ladattava koodi” on REST-arkkitehtuurimallin viimeinen ja ainoa valinnainen rajoite. Se on palvelimen ominaisuus, joka mahdollistaa asiakaspään ohjelman toiminnallisuuden laajentamista palvelimelta ladattavalla koodilla tai ohjelmalla, joka voi olla esimerkiksi Java applet- tai JavaScript-skriptitiedosto. Tämä helpottaa asiakasohjelman toteuttamista, koska osa toiminnallisuuksista on toteutettu näissä koodinpätkissä. Toisaalta ladattava koodi heikentää järjestelmän näkyvyyttä piilottamal-

la osan sen toiminnallisuudesta tämän koodin taakse, minkä vuoksi sen käyttäminen ei ole pakollista. [1, s. 84-85.]

2.2 HTTP-protokolla

HTTP on synkroninen asiakas-palvelin-kommunikointiin perustuva tiedonsiirtoprotokolla. Se on internetin yleisin protokolla, jota esimerkiksi selaimet kuten Firefox, Internet Explorer tai Google Chrome käyttävät internetsivujen selaamiseen. [2, s. 4.] Kuvassa 1 on esimerkki web-palvelusta ja sitä käyttävistä asiakkaista.



Kuva 1. Asiakas-palvelin-malli.

Kuvan oikealla puolella sijaitsee Web-palvelin ja sen lähiverkosta palvelua käyttävä asiakas. Kuvan vasemmalla puolella on kaksi internetin yli palvelua käyttävää asiakasta.

Yksinkertaisuudessaan HTTP toimii seuraavasti: asiakas lähettää pyynnön, joka sisältää HTTP-metodin, pyydettävän resurssin sijainnin, ryhmän otsikkoja ja valinnaisen viestin joka voi sisältää tekstiä tai jopa binääridataa [2, s. 5].

```
GET /time-app/api/hello HTTP/1.1
Host: localhost:8080
Accept: application/json
```

Kuva 2. Esimerkki HTTP GET -pyynnöstä

Asiakasohjelmasta riippuen esimerkiksi selaimet lisäävät automaattisesti kutsuun lisäotsikkoja, jotka kertovat esimerkiksi, mistä asiakasohjelmasta pyyntö lähetettiin. Webselain lähettää kuvassa 2 esitetyn kutsun, jos käyttäjä haluaa päästä käsiksi resurssiin, joka löytyy URL-osoitteesta *localhost:8080/time-app/api/hello* ja odottaa saavansa vastauksen JSON-dokumentin muodossa.

Palvelimen palauttama vastaus on hyvin samankaltainen. Se sisältää lyhyen statuskoodin, valinnaisia otsikkoja ja mahdollisen vastausviestin.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
      "message" : "hello"
}
```

Kuva 3. Esimerkkivastaus kuvan 2 GET-pyyntöön

Kuvan 3 esittämän vastauksen statuskoodi on 200 ja sen viesti “OK”. Tämä koodi tarkoittaa, että pyyntö vastaanotettiin ja prosessoitiin onnistuneesti palauttaen asiakkaalle pyydetyn resurssin. HTTP sisältää suuren määrän statuskoodeja, jotka kertovat pyynnön onnistumisesta tai virheestä. Tämä vastaus sisältää myös viestin, jonka asiakasohjelma osaa tulkita JSON-dokumentiksi Content-Type-otsikon avulla.

HTTP-protokollaan kuuluu rajattu määrä operationaalisia metodeja. Yleisimmät näistä ovat taulukossa 1 esitetyt GET-, PUT-, DELETE-, POST- ja PATCH-metodit.

Taulukko 1. HTTP-metodit ja niiden selitykset

HTTP metodi	Selite
GET	GET on pelkästään tiedon lukemiseen tarkoitettu metodi. Sitä käytetään jonkin tietyn resurssin hakemiseen palvelimelta. Tämä operaatio on idempotentti, joka tarkoittaa että vaikka saman GET-kutsun tekisi monta kertaa, lopputulos on aina sama. Operaatio ei saa myöskään muuttaa resurssia millään tavalla. Siksi sitä kutsutaan myös “turvalliseksi” operaatioksi. [2, s. 8.]
PUT	PUT-operaatiossa palvelinta pyydetään tallentamaan pyynnön sisältämä resurssi samaan osoitteeseen, mihin pyyntö oli tehty. Tämä operaatio on myös idempotentti, koska pyynnön tekijä tietää luotavan tai muokattavan resurssin identiteetin, jolloin saman kutsun tekeminen monta kertaa peräkkäin pitäisi aina johtaa samaan lopputulokseen. [2, s. 8.]
DELETE	DELETE on tarkoitettu nimensä mukaisesti resurssien poistamiseen palvelimelta. Aivan kuten edelliset operaatiot, sekin on myös idempotentti. [2, s. 8.]
POST	POST on HTTP-protokollan ainoa operaatio, joka ei ole idempotentti tai turvallinen. Jokainen pyyntö saa muokata palvelua omalla tavallaan. POST-operaatiota käytetään monesti resurssien luomiseen silloin, kun pyynnön tekijä ei määrittele sen identiteettiä vaan antaa palvelimen luoda sen. [2, s. 8.]
PATCH	PATCH-operaatio on ehdotettu uusi standardi HTTP-protokollalle resurssien päivittämiseen. Se ei ole virallisesti osa standardia, mutta sitä käytetään joissakin web-rajapinnoissa sen hyödyllisyyden takia. PATCH-operaation ideana on päivittää jonkin resurssin tila lähettämällä vain muutettavat attribuutit kutsun mukana. Ilman PATCH-operaatiota resurssin päivitys täytyy tehdä PUT-kutsussa, jonka pitää sisältää muokattavan resurssin tiedot kokonaisuudessaan.
HEAD	HEAD on käytännössä sama operaatio kuin GET, mutta se palauttaa vastauksessa pelkästään otsikot ja statuskoodin [2, s. 8].
OPTIONS	OPTIONS-operaatiota käytetään, kun haetaan tietoa tavoista, miten resurssia voidaan operoida edellä mainituilla metodeilla. Se auttaa asiakasohjelmaa määrittelemään palvelimen ja resurssin valmiuksia. [2, s. 8.]

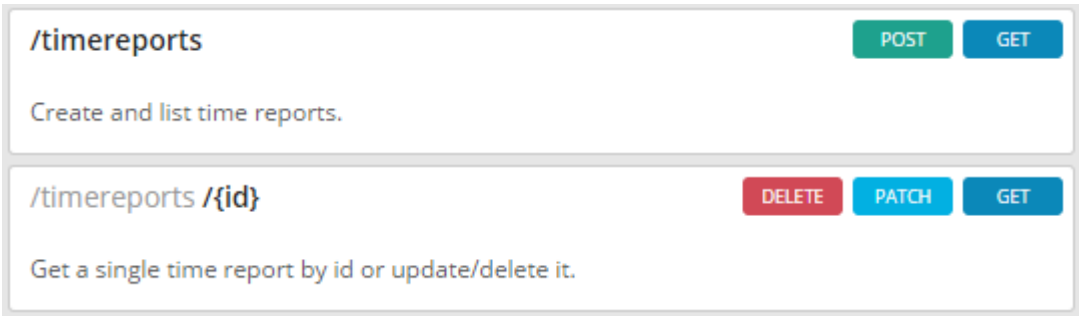
HTTP sisältää edellä mainittujen lisäksi joukon muita operaatioita kuten TRACE tai CONNECT, mutta ne eivät ole olennaisia REST-rajapinnan toteutuksessa [2, s. 8].

Kun palvelin on käsitellyt asiakkaan lähettämän kutsun, antaa se vastauksen mukana niin kutsutun statuskoodin, joka informoi asiakasohjelmaa, kuinka kutsun käsittely onnistui. Usein asiakkaalle annetaan koodi 200, jos kaikki sujui hyvin, 404 jos haettua resurssia ei löytynyt tai 500, jos palvelimen päässä tapahtui vakava virhe. Vastauskoodin tyyppi selviää koodin ensimmäisestä numerosta. 2xx-alkuiset numerot ilmoittavat, että pyyntö käsiteltiin onnistuneesti, 3xx-alkuiset koodit kertovat, että haettu resurssi löytyy jostain muualta. 4xx-alkuinen koodi tarkoittaa, että asiakasohjelma on tehnyt

virheellisen kutsun. 5xx-alkuisen koodin sattuessa palvelimen päässä on tapahtunut vakava virhe eikä pyyntöä voitu käsitellä onnistuneesti loppuun.

2.3 HTTP-protokolla REST-arkkitehtuurimallin toteuttajana

HTTP-protokolla noudattaa REST-arkkitehtuurimallia täyttämällä sen kaikki asettamat rajoitteet. Se perustuu asiakas-palvelin-malliin ja oikein käytettynä tarjoaa yhtenäisen rajapinnan määrittelemällä tarkan rakenteen ja syntaksin viesteille hyväksikäyttämällä protokollan tarjoamia operationaalisia metodeja, vastauskoodeja ja URL-osoitteita. Se mahdollistaa välimuistin käyttämisen viesteihin lisättävien otsikoiden avulla. Tilattomuus-rajoitteen noudattaminen riippuu täysin palvelimen toteutuksesta, joten tiedonsiirtoprotokollana HTTP ei ota siihen kantaa.



Kuva 4. Yleisnäkymä tuntikirjausresurssin dokumentaatiosta.

Kuvassa 4 esitetään tyypillisen REST-resurssin URL-osoitteet ja niiden hyväksymät HTTP-metodit. Ylemmän osoitteen kuvastamaa resurssia kutsutaan usein kokoelmaksi (collection) ja alempaa yksittäiseksi resurssiksi (single resource). Tuntikirjausten luominen onnistuu tekemällä ylempään osoitteeseen POST-tyyppinen HTTP-kutsu, joka sisältää tuntiraportin tiedot. Vastauksena palvelin palauttaa yleensä luodun tuntikirjauksen tunnisteen (id). Kun tiedossa on tuntikirjauksen tunnus, voidaan yksittäisen tuntikirjauksen tiedot hakea palvelimelta tekemällä GET-kutsu alempaan osoitteeseen, jossa ”{id}” korvataan tuntikirjauksen tunnisteella. Tuntikirjausta voidaan myös muokata tai sen voi poistaa käyttämällä PATCH- ja DELETE-kutsuja. Tekemällä GET-kutsun ylempään URL-osoitteeseen voidaan hakea listaus tuntikirjauksista. Listausresurssit tukevat usein suodattimia listattavien resurssien karsimiseksi. HTTP-kutsuissa se onnistuu helposti käyttämällä hyväksi osoitteen perään lisättäviä query-parametreja. Kuvan 3

esimerkissä suodattimia voisivat olla esimerkiksi tuntikirjauksen tekijä tai aikaväli, jonka sisälle sijoittuvat tuntiraportit halutaan hakea.

3 PlanMill-sovellus

3.1 PlanMill Open API -rajapinta

PlanMill Open API on ilmainen web-rajapinta PlanMillin CRM-, Project- ja ERP-ohjelmistojen asiakkaille. Se tukee CRUD-operaatioita järjestelmän resursseihin kuten asiakastileihin tai tuntiraportteihin. Asiakkaat ovat käyttäneet rajapintaa suurimmaksi osaksi sovellusintegraatioissa. Esimerkiksi eräs asiakas on rakentanut integraation, jossa automatisoidaan PlanMill-sovelluksen projektitietojen kirjaus Atlassian JIRA -tehtävienhallintasovellukseen ja JIRAn tuntikirjauksien synkronisointi PlanMill-sovellukseen.

Asiakkailta saadun palautteen pohjalta on kuitenkin todettu, että rajapinnassa on kehittämisen tarvetta, eikä se myöskään noudata yleisiä REST-standardeja. Open API sisältää REST-palvelun, mutta nimestään huolimatta se ei noudata REST-arkkitehtuurimallin periaatteita. Kaikki rajapintakutsut tehdään samaan URL-osoitteeseen ja käsiteltävän resurssin tyyppi ja vastauksena palautettavan dokumentin muoto annetaan HTTP query-parametrina URL-osoitteen lopussa. Insert- ja update-kutsuissa lisättävän resurssin tiedot lisätään myös kutsun URL-parametreihin. Autentikaatio tapahtuu lisäämällä jokaiseen API-kutsuun käyttäjän tunnistenumero ja autentikaatioavain. Rajapinta hyväksyy vain GET- ja POST-kutsuja, mutta käytännössä metodilla ei ole muuta merkitystä, kuin että POST-kutsuissa parametrit voidaan sisällyttää form-parametreihin. Seuraavana on esimerkki kutsusta, jolla haetaan yhden asiakastilin tiedot:

https://[server]/[instance]/services/rest?format=rest&method=account.get&account.id=815867&userid=12345&authkey=531adbb3022a96e537b4e2a5289e128e

Kutsun format-parametriksi on määritetty ”rest”- joka kertoo palvelimelle, että sen pitää palauttaa vastauksensa kuvassa 4 esitetyssä formaatissa. Method-parametrin arvo ”account.get” määrittää kutsun operaation, joka on tässä tapauksessa yhden asiakasti-

lin noutaminen. Parametrissa ”account.id” on määritetty sen asiakastilin tunniste, jonka tiedot haetaan. Parametrit ”userid” ja ”authkey” sisältävät rajapinnan käyttäjän autentikaatioon tarvittavat tiedot: käyttäjän tunnisteen ja autentikaatioavaimen. Esimerkkivastaus edelliseen kutsuun esitetään koodiesimerkissä 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok" method="account.get">
    <accounts>
        <account id="815867">
            <Id>815867</Id>
            <Muokkaa>ui-icon-pencil</Muokkaa>
            <attachments/>
            <Access/>
            <ActiveContacts>33</ActiveContacts>
            <Account.Name>Acme Oy</Account.Name>
            <Address1>Esimerkkitie 12&lt;br/&gt;02200 Espoo</Address1>
            <Location1.CountryId>Finland</Location1.CountryId>
            <Account.Phone>+358 4 4567 6700</Account.Phone>
            <Account.Website>http://www.acme.com</Account.Website>
            <Account.Type>Asiakas</Account.Type>
            <Person.LastName>Hoikkanen, Seppo</Person.LastName>
            <Account.Email>@acme.com</Account.Email>
            <PassiveStyle>color: #659D32; ;</PassiveStyle>
            <Account.SocialMedia4/>
            <Account.SocialMedia2/>
            <Account.SocialMedia3/>
            <SupplierAccount.Id/>
            <Location2.Id>893675</Location2.Id>
            <Location1.Id>815868</Location1.Id>
            <ParentAccount.Id/>
            <Person2.Id>921831</Person2.Id>
            <Person1.Id>26394</Person1.Id>
            <Person.Id>24715</Person.Id>
        </account>
    </accounts>
</rsp>
```

Koodiesimerkki 1. Rajapinnan vastaus HTTP-kutsuun, jossa haettiin yhden asiakastilin tiedot.

Koodiesimerkin 1 esittämä XML-dokumentti sisältää yhden asiakastilin tiedot. Dokumentin Muokkaa- ja PassiveStyle-elementeistä huomataan heti, että dokumentti sisältää käyttöliittymään liittyvää informaatiota, jotka eivät web-rajapintojen kannalta ole oleellista tietoa. Account.Type -kentästä nähdään, että rajapinnan palauttama tieto on käännetty järjestelmän formaatista (lukuenumeraatio tässä tapauksessa) valmiiksi käyttäjän omalle kielelle, jonka johdosta rajapinnan yhtenäisyys kärsii. Uutta asiakastiliä luodessa tämä tieto pitää antaa järjestelmän omassa formaatissa eli lukuna.

Seuraavana on esimerkkikutsu, jossa lisätään yksi uusi kontakti, joka sisältää järjestelmän asettamat minimitiedot:

https://[server]/[instance]/services/rest?format=rest&method=contact.insert&userid=12345&authkey=531adbb3022a96e537b4e2a5289e128e?person.lastname=Esimerkki&person.languagecode=fi&person.passive=0&location1.id=54321&location2.countryid=81&accounthasperson.accountid=98765&person.projectwikiusergroupid=56789&return=true

URL-osoite on sama kuin edellisessä kutsussa, mutta luotaessa uutta kontaktia ”method”-parametrin arvoksi pitää laittaa ”contact.insert”. Kaikki luotavaan kontaktiin liittyvä tieto sisällytetään kutsuun query-parametereinä. ”Return”-parametrin arvon ollessa ”true” palvelin sisällyttää kontaktin luomisen jälkeen palautettavaan HTTP-vastaukseen luodun kontaktin tunnisteen.

3.2 Uuden rajapinnan toteutus PlanMill-sovellukseen

Aluksi tutkin, olisiko uusi rajapinta mahdollista toteuttaa vanhan rajapinnan tavoin nykyisen PlanMill-järjestelmän päälle. Nykyinen rajapinta on käytännössä sovelluksessa pyörivä hyvin yksinkertainen Java HTTP Servlet, joka vastaanottaa HTTP-kutsuja ja kutsuu sovelluslogiikkaa kutsuissa annetuilla parametreilla. Servlet palauttaa vastauksena samanlaista tietoa, kuin jos ohjelmaa käytettäisiin selainkäyttöliittymän kautta.

Koska käyttöliittymä rakennetaan suureksi osaksi sovelluksen sisällä, vastaus sisältää käyttöliittymään liittyvää informaatiota, joka olisi uuden rajapinnan kannalta täysin epäolennaista. Myös rajapinnan palauttama tieto suodatetaan järjestelmään tallennettujen käyttäjäparametrien mukaan. Tämä on aiheuttanut joillakin käyttäjillä hämmennystä, kun yksinkertainen kysely palautti tyhjän vastauksen tai vain pienen määrän tietoja, koska osa riveistä oli vaivihkaa suodatettu pois. Esimerkiksi jos käyttäjän projektinäkymän asetuksissa on suodatin, joka näyttää vain 100 ensimmäistä projektia, niin haku, joka yrittää noutaa kaikki projektit, palauttaa vain ensimmäiset 100 tulosta. Tätä varten rajapinnan käyttöä varten on yleensä luotu oma käyttäjänsä, jolta turhat suodattimet on konfiguroitu pois päältä.

Ylimääräisen tiedon suodattaminen ja lopun tiedon muotoilu REST-malliseksi osoittautui hyvin hankalaksi ja työlääksi toteuttaa, joten vanhan rajapinnan hyväksikäyttäminen ei ole pätevä ratkaisu. Toinen vaihtoehto olisi kutsua suoraan sovelluksen bisneslogiik-

kaa, mutta käyttöliittymään liittyvä tieto käsitellään bisneslogiikassa, mikä rikkoo perinteistä MVC-sovellusarkkitehtuuria sitomalla yhteen mallin ja näkymän. Jotta vanha sovelluslogiikka saataisiin taivutettua uutta rajapintaa tukevaksi, jouduttaisiin tekemään paljon muutoksia suureen osaan olemassa olevasta koodimassasta.

Lopulta tulimme tulokseen, että sovellus on liian epäyhteensopiva uuden rajapinnan kanssa, koska sitä varten sovellukseen jouduttaisiin tekemään suuria rakenteellisia muutoksia rajapinnan toteuttamiseksi.

4 Uuden arkkitehtuurin tarpeiden kartoitus

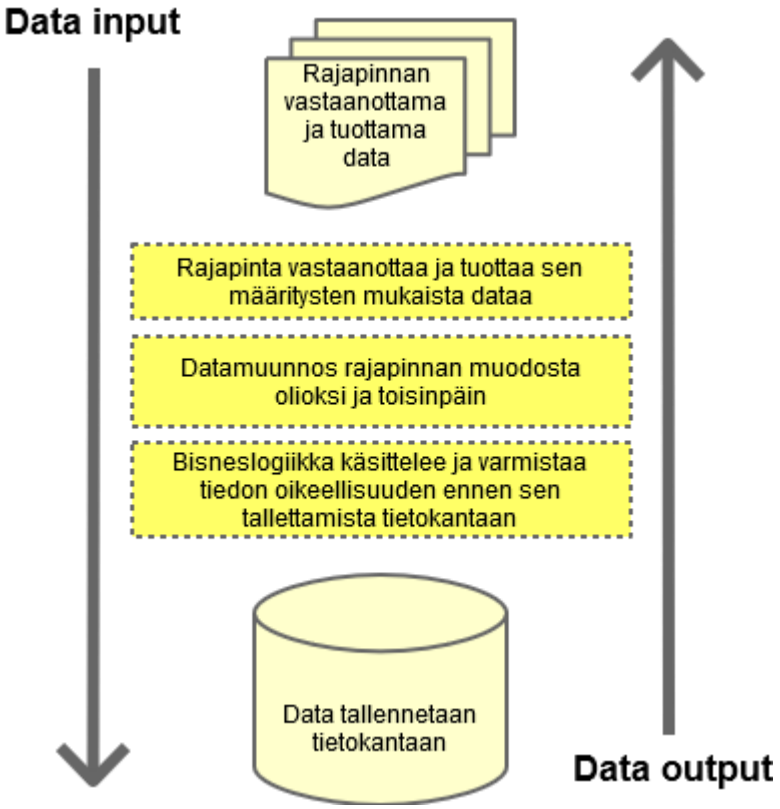
Tässä vaiheessa uuden rajapinnan toteuttaminen muuttui paljon laajemmaksi arkkitehtuuriprojektiksi. Koko nykyisen PlanMill-ohjelmiston palvelujen kattavan rajapinnan tekeminen veisi huomattavan paljon aikaa, joten keskityn tässä työssä vain yhteen pienempään osa-alueeseen eli ohjelmaan, joka mahdollistaa tuntikirjausten käsittelyn. Insinöörityön tuloksena olisi tarkoitus toteuttaa uuden arkkitehtuurin päälle esittelyversio uudesta PlanMill TimeAppista, joka mahdollistaa tuntiraporttien selaamisen, kirjaamisen ja muokkauksen.

Nykyistä PlanMill-ohjelmistoa ajetaan tuotannossa Tomcat-web-palvelimella. Tästä syystä on luonteva ratkaisu kehittää uusi arkkitehtuuri toimimaan samalla alustalla. Tämä tarkoittaa myös sitä, että uusi arkkitehtuuri tulee pohjautumaan samaan ohjelmointikieleen, eli Javaan.

Kaiken tiedon lähettäminen ja vastaanottaminen tulee tapahtumaan yhden rajapinnan kautta, jonka pitää noudattaa REST-arkkitehtuurimallia. Tähän ei kuitenkaan lasketa mukaan mahdollisia sovellusintegraatioita, jotka jäävät tämän insinöörityön aiheen ulkopuolelle.

Sovelluksen pitää pystyä tallettamaan pysyvää tietoa jonnekin. Luonnollinen ratkaisu tähän on käyttää tietokantaa, jota nykyinen PlanMill-sovellus myös käyttää. Se ei kuitenkaan sisällä minkäänlaista olio-relaatio-muunnosta tietokantatietueiden ja Java-olioden välillä, mutta uuteen arkkitehtuuriin se on tarkoitus toteuttaa.

Rajapinnan käsittelemä tieto ei ole sellaisenaan yhteensopiva sen muodon kanssa, millaisena sitä käsitellään järjestelmän sisällä. Tätä varten pitää olla jonkinlainen komponentti, joka hoitaa rajapinnan vastaanottamien dokumenttien muuntamisen Java-olioiksi ja toisinpäin.



Kuva 5. Arkkitehtuurin vaatimusten pohjalta laadittu tietovirtadiagrammi.

Jäljelle jää enää tiedon käsittely. Kaikki sovelluksen vastaanottama tieto pitää todeta oikeelliseksi ensin varmistamalla, että sen rakenne on rajapinnan määrittämän mallin mukainen. Sen jälkeen tieto validoidaan siihen koskevia bisnessääntöjä vasten, jonka jälkeen tieto voidaan todeta oikeelliseksi ja tallentaa tietokantaan. Edellä mainittujen vaatimusten perusteella laadin kuvan 6 tapaisen kaavion, joka kuvaa sovelluksen toimintaa tiedon kulkiessa järjestelmään ja siitä ulos.

Tämän työn osalta arkkitehtuurisuunnittelun ulkopuolelle jäävät nykyisen PlanMill-järjestelmän yksi tärkeimmistä komponenteista, joka on monipuolisen konfiguraation ja kustomoinnin mahdollistava parametrijärjestelmä. Pois jää myös autentikaatio, jolla

tunnistetaan järjestelmän käyttäjän identiteetti ja.varmistetaan, että käyttäjä pääsee käsiksi resursseihin, joihin hänellä on oikeus.

5 Arkkitehtuuriin sisältyvät teknologiat

5.1 Apache Tomcat

Tomcat on Apache Software Foundationin vapaan lähdekoodin web-palvelin ja servlet-säiliö. Tomcat toteuttaa Oraclen (alun perin Sun Microsystemsin) JSP- ja Servlet-tekniikoita, jotka mahdollistavat Java-pohjaisten web-sovellusten ajamisen. Itse Tomcat on myös toteutettu Javalla ja sitä suoritetaan Javan virtuaalikoneen sisällä. [3.]

5.2 JAX-RS ja Jersey Framework

JAX-RS on Oraclen (alun perin Sun Microsystemsin) kehittämä web-rajapinta, jonka avulla voi toteuttaa REST-arkkitehtuurimallin mukaisia verkkopalveluita Javalla. JAX-RS on ollut virallinen osa Java EE -ohelmistokehitysalustaa sen kuudennesta versiosta lähtien. JAX-RS käyttää hyväksi Javan annotaatioita luokkien muuttamisessa REST-arkkitehtuurimallin mukaisiksi resursseiksi. JAX-RS:stä julkaistiin versio 2.0 vuoden 2013 toukokuussa. Jersey, jota käytän tässä projektissa, on Oraclen oma vapaan lähdekoodin mallitoteutus JAX-RS-standardista. [4.]

5.3 Jackson JSON-prosessori

Jackson on monikäyttöinen Java-kirjasto JSON-formaatin prosessoimiseen. Se tarjoaa pääsääntöisesti kolme eri tapaa JSON-datan lukemiseen ja kirjoittamiseen. Ensimmäisenä on nopea Streaming API, joka lukee tai kirjoittaa tiedon järjestyksessä käyttäen samalla mahdollisimman vähän muistia. Toinen käsittelytapa on luoda ja lukea dataa helposti muokattavaan puurakenteeseen, joka sisältää JsonNode-solmuja. Kolmas ja yleisesti kätevin tapa prosessoida JSON-tietoa on käyttää Jacksonin Data Bindingia. Sen avulla tieto muunnetaan saumattomasti JSONista JavaBean-olioihin ja toisin päin. [5.]

5.4 JPA ja Hibernate ORM

JPA on ORM-rajapinta, joka kehitettiin tietokannoissa sijaitsevan tiedon käsittelyyn Java-ohjelmassa. Sen avulla voidaan automatisoida muunnokset olio- ja relaatiomallien välillä ja toteuttaa olioiden pysyvä tallennus tietokantaan. Relaatiotieto kuvataan kevyinä JavaBean-olioina, joiden muuttujat kuvaavat jonkin tietokantataulun sarakkeita. Yksi tällaisen olion ilmentymä vastaa yhtä tietokantataulun riviä. Entiteettien väliset riippuvuudet määritellään olio-relaatio-metatiedolla, jonka voi sijoittaa suoraan entiteetteihin Javan annotaatioilla tai erillisiin xml-tiedostoihin. JPA tarjoaa myös oman SQL-syntaksia muistuttavan JPQL-kyselykielen, jonka kyselyt kohdistetaan tietokantataulujen sijaan entiteetteihin.

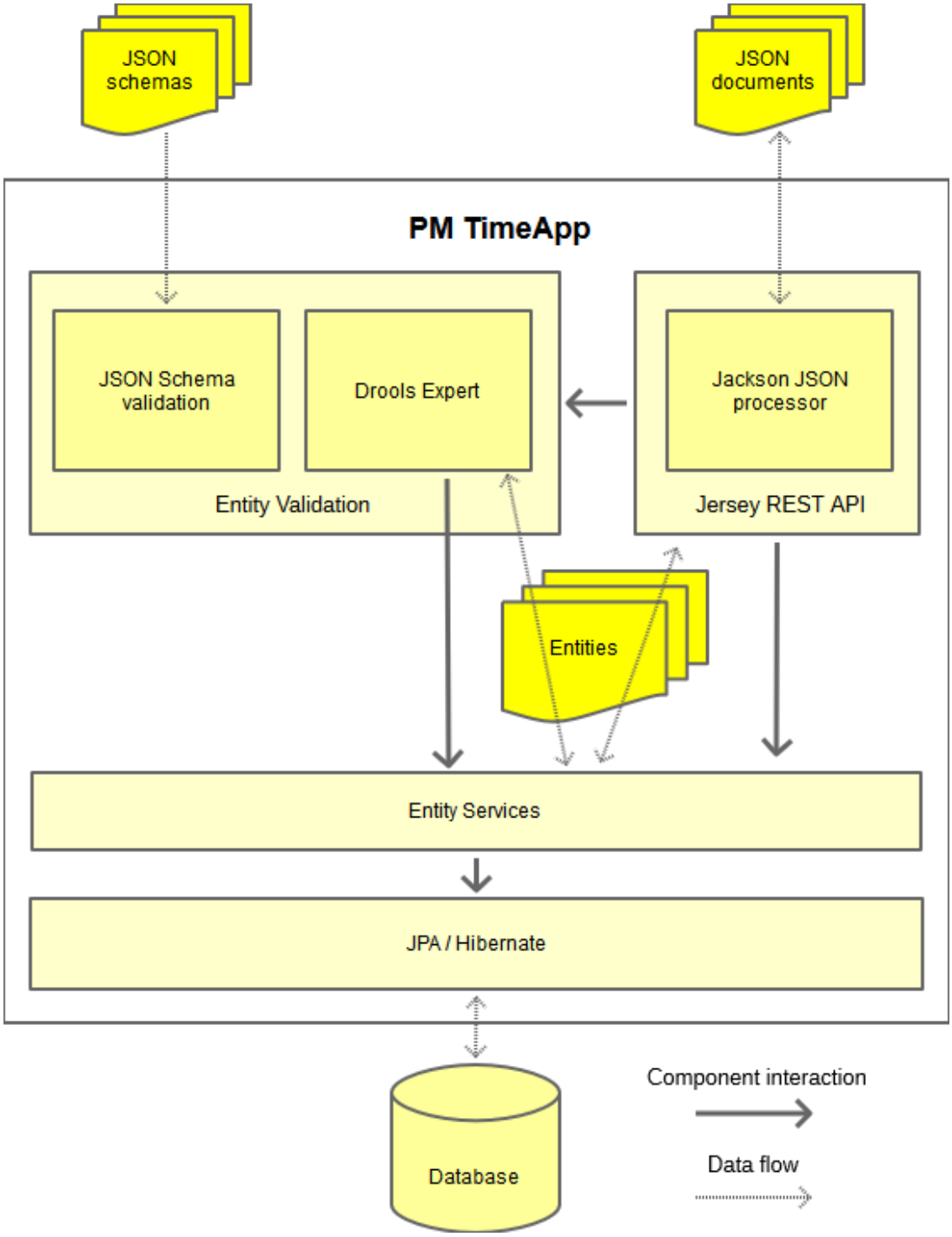
Hibernate ORM on Red Hatin kehittämä vapaan lähdekoodin aliohjelmakirjasto, joka toimii samalla tavalla kuin JPA. Hibernaten käyttö voidaan toteuttaa JPA-rajapinnan kautta tai käyttäen Hibernaten omia luokkia ja annotaatioita. Hibernate tarjoaa myös oman HQL-kyselykielen, josta JPQL-kieli alun perin kehitettiin. Hibernate sisältää monia hyödyllisiä lisäominaisuuksia kuten entiteettien auditoinnin ja versioinnin Hibernate Enversillä tai Apache Lucenea hyväksi käyttävän Hibernate Searchin, joka mahdollistaa full-text-hakujen tekemisen entiteetteihin. [6;7.]

5.5 JBoss Drools

JBoss Drools on bisneslogiikkaintegraatioalusta (Business Logic Integration Platform, BLiP), jonka ydin on Rete-hahmonsovitusalgoritmiin perustuva sääntömoottori. Drools tarjoaa kokonaisvaltaisen ratkaisun bisnessääntöjen luomiseen, soveltamiseen ja hallinnointiin. Droolsista on olemassa useampi versio, joista JBoss Entreprise BRMS ja JBoss Rules ovat Droolsin tuotteistettuja versioita, jotka sisältävät enterprise-tason käyttötuen. Drools itsessään on JBoss-yhteisön vapaan lähdekoodin projekti, joka sisältää useita moduuleja. Yksi näistä on Drools Expert, joka sisältää itse sääntömoottorin. Muita moduuleja ovat muun muassa sääntöjenhallintajärjestelmä Drools Guvnor, prosessienhallintajärjestelmä jBMN ja tapahtumalähtöisen bisnesvalidoinnin mahdollistava Drools Fusion. [8.]

6 Arkkitehtuurin suunnittelu ja toteuttaminen

Seuraavaksi käydään läpi suunnittelun tuloksena syntynyt näkemys uuden järjestelmän kokonaisarkkitehtuurista.



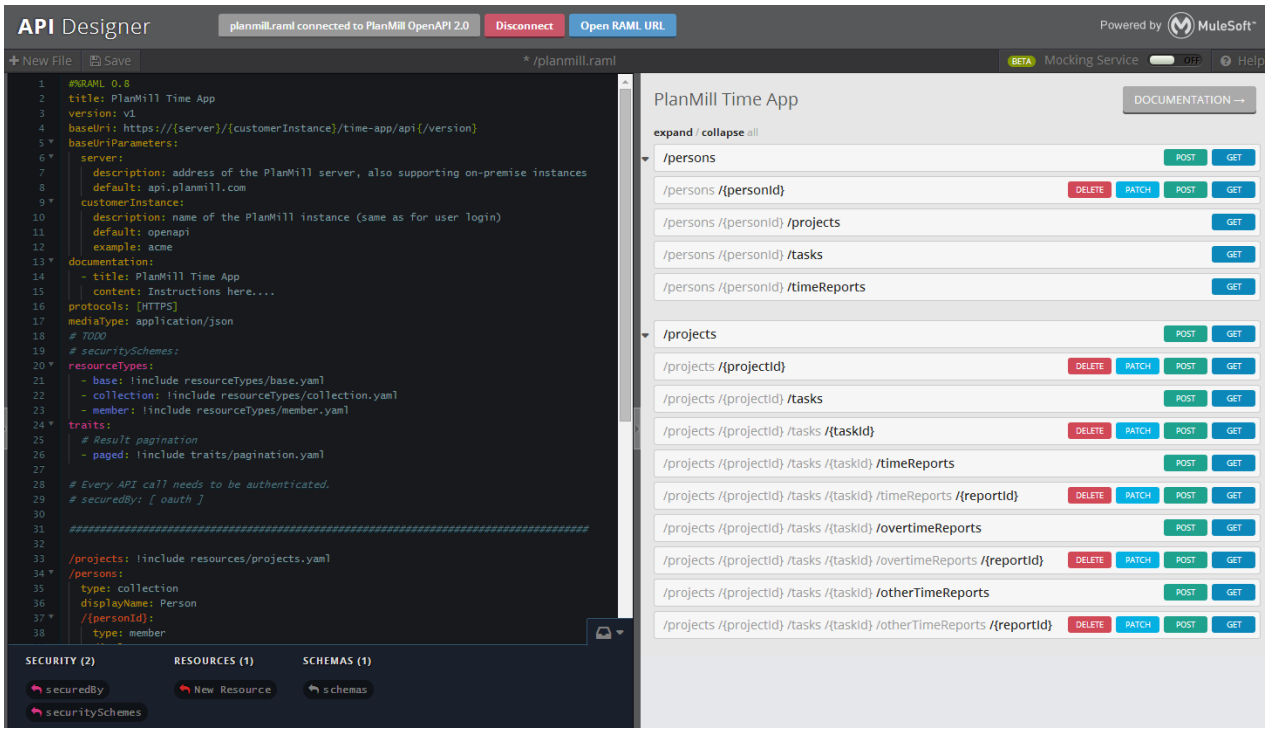
Kuva 6. Uuden arkkitehtuurin pääkomponentit.

Kuva 7 esittää uutta sovellusarkkitehtuuria. Se koostuu neljästä pääkomponentista, jotka ovat REST-rajapinta (Jersey REST API), entiteettivalidaatio (Entity Validation), entiteettipalvelut (Entity Services) ja tietokantarajapinta (JPA/Hibernate). REST-rajapinta on vastuussa HTTP-kutsujen käsittelystä ja niiden sisältämän tiedon muuttamisesta JSON-tekstidokumenteista (JSON document) oliomalleiksi ja toisinpäin. Rajapinta käyttää entiteettivalidointia tarkistamaan rajapintaan tuodun tiedon oikeellisuuden hyväksikäyttäen JSON-skeemoja (JSON schemas) ja Drools Expert -sääntömoottoria. Drools ja REST-rajapinta käyttävät entiteettipalvelukerrosta tiedon tallentamiseen ja hakemiseen tietokannasta (Database). Entiteeteillä (Entities) tarkoitetaan tässä kontekstissa tietokannan tietueita, jotka on muunnettu ORM-mallin mukaisiksi java-olioksi.

6.1 REST-rajapinta

Rajapinnalla täytyy olla spesifikaatio. REST-rajapinnan tapauksessa se sisältää yleensä vähintään listauksen kaikista rajapinnan resursseista ja niiden URL-osoitteet, tiedot rajapinnan tukemista HTTP-metodeista, vastaanottavasta ja/tai palauttavasta datasta ja autentikaatiometodeista. On olemassa useita työkaluja, jotka auttavat rajapintaspesifikaation luomisessa ja dokumentaatiossa. Näistä esimerkiksi Anypoint API Platform ja Apiary ovat pilvipalveluita, jotka tarjoavat spesifikaation suunnittelun lisäksi automaattisen dokumentaation generoinnin, testaustyökalut ja omat kielensä rajapinnan kuvaamiseen. Toinen vaihtoehto on käyttää ohjelmia tai kirjastoja, kuten I/0 Docs, Swagger tai Enunciate, jotka ovat ominaisuuksiltaan hieman rajoittuneempia kuin edellä mainitut palvelut.

Päätimme, että rajapinnan kuvaus tehdään Mulesoft Anypoint API Platormia käyttäen. Rajapintakuvaus kirjoitetaan RAML-kuvauskielellä, joka pohjautuu YAML-merkintäkieleen. API Platform tarjoaa oman online-editorin spesifikaation kirjoittamiseen, ja se kykenee luomaan rajapintadokumentaatiota lennosta. Myös rajapinnan matkiminen on mahdollista beta-vaiheessa olevan mock-palvelun avulla. Palveluun voi liittää myös GitHub-tilin, joka mahdollistaa spesifikaation ja testien helpon versioinnin.



Kuva 7. Mulesoft Anypoint API Platform API Designer.

Kuvan 8 vasemmalla puolella näkyy RAML-konsoli, johon kirjoitetaan rajapinnan spesifikaatio RAML-kuvauskielellä. Speksiä muokattaessa ohjelma generoi jokaisen muutoksen jälkeen automaattisesti dokumentaation oikealla puolelle. Dokumentaation jokaisen rivin vasemmalla puolella on resurssin suhteellinen URL-osoite rajapinnan juuresta ja oikealla puollella värillisinä palikoina kerrottu, mitä HTTP-metodeja kyseiset resurssit tukevat. Aaltosulkeilla merkityt segmentit kuvastavat resurssin yksilöllistä tunnistetta. Esimerkiksi *projects/{projectId}* voidaan tulkita *projects/1234*, jolloin viitataan projektiresurssiin, jonka tunniste on 1234.

Tähän projektiin ei otettu mukaan käyttäjien autentikaatiota. Sen toteuttaminen veisi mahdollisesti huomattavan paljon aikaa, eikä se ole hirveän oleellinen projektin kannalta. Käytönnössä kuka tai mikä tahansa voi tehdä HTTP-kutsuja kaikkiin resursseihin ja muokata, luoda ja poistaa niitä mielensä mukaan. Normaalisti kaikissa web-apeissa on autentikaatio, joka pitää huolen, että vain ne käyttäjät, joilla on resursseihin oikeus, pääsevät niihin käsiksi.

Uusi REST-rajapinta toteutetaan käyttämällä Oraclen Jersey-kirjastoa. Se sijoitetaan Javan Servlet-säiliöön, jolloin sitä voi ajaa millä tahansa web-palvelimella, joka tukee

Java EE-arkkitehtuurin web-komponentteja. Tässä tapauksessa palvelimena toimii Apache Tomcat. Servletin asetuksiin laitetaan URL-osoite, jotta palvelin osaa ohjata siihen tehdyt pyynnöt Jerseylle. Konfiguraation voi tehdä perinteisesti lisäämällä servletin asetukset Servlet Containerin web.xml-tiedostoon tai annotoimalla luokan, joka periytyy javax.ws.rs.core.Application-luokasta tai Jerseyn tapauksessa org.glassfish.jersey.server.ResourceConfig-luokasta, joka itsessään periytyy Applicationista ja sisältää Jerseyn konfiguraatiota helpottavia metodeja.

Kaikki rajapintaan kohdistetut kutsut ohjataan yhden juuriresurssiluokan kautta kutsun URLin ja HTTP-kutsun operationaalisen metodin avulla oikeaan resurssiin. Koodiesimerkissä 2 on esitelty osa Persons-resurssin Java-luokasta. Resurssiluokkien metodit sisältävät annotaatioita, jotka kertovat, mitkä kutsut niihin pitää ohjata, sekä ne mediatyypit, mitä rajapinta vastaanottaa kutsuissa ja palauttaa vastauksissa.

```
public class PersonsResource {

    private static final Logger log = LogManager.getLogger(PersonsResource.class);
    private UriInfo             uriInfo;

    public PersonsResource(UriInfo uriInfo) {
        this.uriInfo = uriInfo;
    }

    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Response getPersons(@BeanParam PaginationBean pagination) throws JsonProcessingException {
        List<Person> persons = PersonService.getAllPersons(pagination.getFrom(), pagination.getPerPage());
        if (persons.isEmpty()) {
            return noContentResponse();
        }
        String json = SerializationUtil.getWriter().writeValueAsString(persons);
        return okResponse(json);
    }

    @POST
    @Consumes({MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_JSON})
    public Response createNewPerson(Person person) {
        List<String> validationErrors = EntityValidator.validateEntity(person);
        if (!validationErrors.isEmpty()) {
            return clientErrorResponse(new ErrorMessage(400, "validation", validationErrors));
        }
        int id = PersonService.createNewPerson(person);
        URI resourceURI = uriInfo.getAbsolutePathBuilder().path(String.valueOf(id)).build();
        log.info("Created a new person " + resourceURI);
        return createdResponse(null, resourceURI);
    }

    @GET
    @Path("/{personId}")
    @Produces({MediaType.APPLICATION_JSON})
    public Response getPerson(@PathParam("personId") int personId) throws JsonProcessingException {
        Person person = PersonService.getPersonForId(personId);
        if (person == null) {
            return notFoundResponse(new ErrorMessage(404, "not found", "Person with id " + personId + " not found"));
        }
        String json = SerializationUtil.getWriter().writeValueAsString(person);
        return okResponse(json);
    }
```

Koodiesimerkki 2. Osa Persons-resurssin Java-luokasta.

Kaikki metodit palauttavat Response-olion, joka sisältää kaikki HTTP-vastauksen tiedot mukaan lukien rajapinnasta haetun resurssin. Palautettava resurssi, eli yleensä käytännössä jonkin entiteettiluokan ilmentymä, annetaan sellaisenaan Jerseyn Response-luokan ilmentymälle, jolloin Jerseyhyn integroitu JSON-prosessori serialisoi sen automaattisesti JSON-muotoon.

6.2 JSON-datan prosessointi

JSON-tiedon käsittely tapahtuu täysin automaattisesti, koska Jackson on integroitu suoraan Jerseyhyn. POST-, PUT- tai PATCH-kutsuissa lähetetty JSON-data deserialisoidaan Javabean-olioksi ja injektoidaan resurssiluokan metodin parametriksi. Vastaavasti, kun Javabean-oliot lisätään GET-kutsun palauttamaan vastaukseen, Jackson serialisoi ne automaattisesti JSON-muotoon.

Melkein jokainen resurssi sisältää viitteitä johonkin toiseen resurssiin, esimerkiksi asiakastiliin liittyy aina kontakti. Ohjelman sisällä nämä viitteet käsitellään Java-olioiden ilmentyminä, kun taas rajapinta esittää viitteet numeerisina tunnisteina (entiteetin id-arvo). Jotta Jackson osaisi deserialisoida POST-kutsuissa annetut numeeriset toisiin entiteetteihin viittaavat id-arvot olioiksi ja GET-kutsuissa Java-luokkien olioviittaukset numeerisiksi id:ksi, joudutaan entiteettiluokkiin lisäämään @JsonIdentityInfo-annotaatio. Siinä on määritelty, että entiteettien “id”-muuttuja sisältää jokaiselle uniikin identiteetin. EntityIdResolver-luokka hoitaa id:n ja scopen sisältämän luokan avulla oikean entiteetin lataamisen tietokannasta.

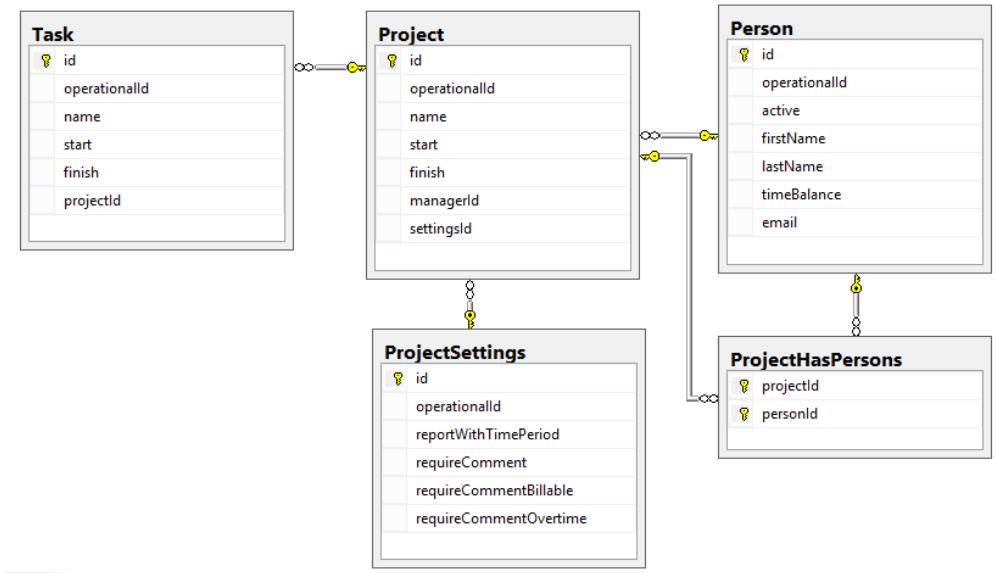
6.3 Tietokanta ja ORM

Nykyinen PlanMill-järjestelmä ei sisällä persistence layeriä tietokannan ja bisneslogiikan välillä, vaan bisneslogiikka tekee SQL-kyselyitä suoraan JDBC-ajurin kautta tietokantaan. Tästä syystä ohjelman koodissa ja parametreissa on paljon tietokannasta riippuvia muuttujia ja vakioita, mikä tietokantatoteutusta vaihtaessa hankaloittaa muutosprosessia huomattavasti.

Uudessa arkkitehtuurissa ideana on käyttää Javan Persistence APIa eli JPAta tietokantataulujen kuvaamisessa Java-olioiksi. JPA:n toteuttavaksi kirjastoksi valittiin Hibernate

ORM. Koska Hibernate toteuttaa JPA:n rajapintaa, pitäisi ORM-kirjaston vaihtaminen toiseen JPA-rajapinnan toteuttajaan onnistua minimaalisella koodin muutoksilla. Myös alla olevan tietokantamoottori on mahdollista vaihtaa helposti Hibernaten abstrahoinnin ansiosta. Tietokanta-ajurina käytetään jTDS JDBC-ajuria ja tietokantayhteyksien hallintaan c3p0-kirjastoa, joista molemmat ovat vapaan lähdekoodin Java-kirjastoja.

Tietokantataulujen kuvaaminen Java-luokiksi on helppoa. Entiteettiluokkien konfiguroiminen voidaan tehdä xml-tiedostoilla tai entiteettiluokissa sijaitsevilla Java-annotaatiolla. Jälkimmäisessä tapauksessa yksinkertaisin toimiva konfiguraatio saadaan luomalla jokaista tietokantataulua varten oma JavaBean-olio, joka sisältää luokkamuuttujat ja niiden getterit ja setterit ja jonka luokkamääritys annotoidaan @javax.persistence.Entity annotaatiolla, kuten kuvasta 8 näkyy. Näin jokainen luokan julkinen tai gettereillä ja settereillä käsiksi päästävä muuttuja tulkitaan tietokantataulun kentiksi. Ohessa on esimerkki Project-tietokantataulusta ja sitä vastaavasta entiteettiluokasta.

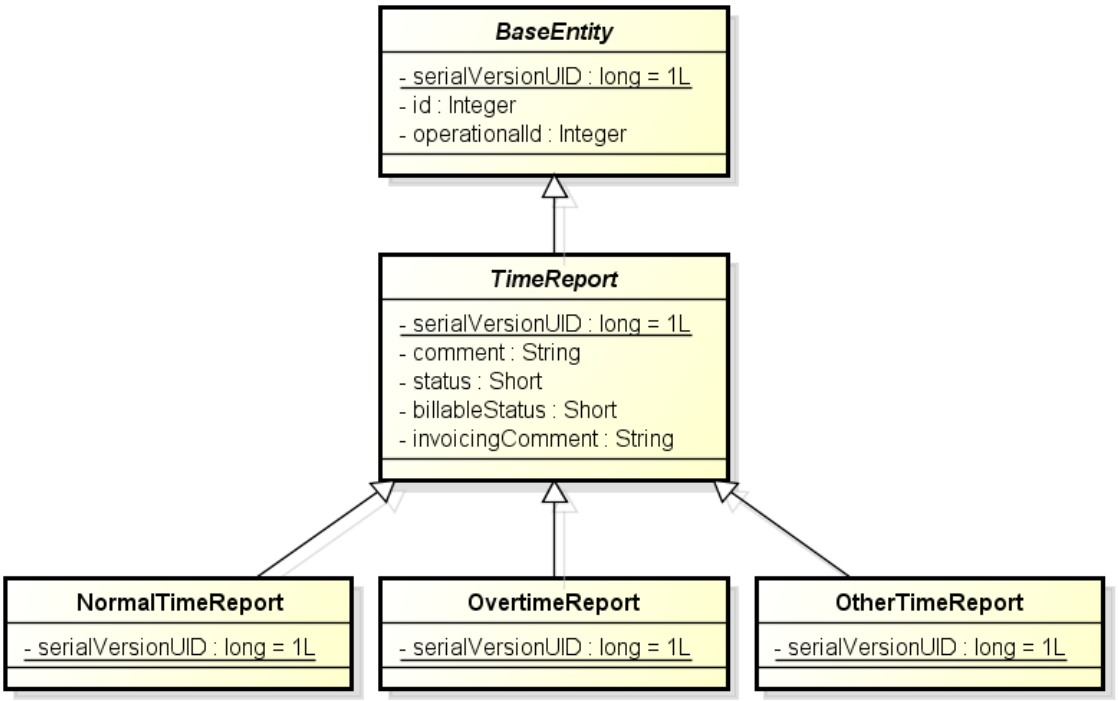



```
23  @Entity
24  @Audited
25  public class Project extends BaseEntity {
26
27      private static final long serialVersionUID = 1L;
28
29      @Column
30      private String            name;
31
32      @Column
33      @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime",
34              parameters = {@Parameter(name = "databaseZone", value = "UTC"),
35              @Parameter(name = "javaZone", value = "UTC")})
36      private DateTime          start;
37
38      @Column
39      @Type(type = "org.jadira.usertype.dateandtime.joda.PersistentDateTime",
40              parameters = {@Parameter(name = "databaseZone", value = "UTC"),
41              @Parameter(name = "javaZone", value = "UTC")})
42      private DateTime          finish;
43
44      @ManyToOne(fetch = FetchType.LAZY)
45      @JoinColumn(name = "managerId")
46      private Person            manager;
47
48      @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
49      @JoinColumn(name = "settingsId")
50      private ProjectSettings   settings;
51
52      @ManyToMany(fetch = FetchType.LAZY)
53      @JoinTable(name = "ProjectHasPersons",
54              joinColumns = @JoinColumn(name = "projectId", referencedColumnName = "id"),
55              inverseJoinColumns = @JoinColumn(name = "personId", referencedColumnName = "id"))
56      private Set<Person>       teamMembers;
```

Kuva 8. Ylempänä diagrammi projektin Project-tietokantataulusta yhteyksineen muihin tauluihin. Alempana kuva Project-taulua vastaavasta entiteettiluokasta

Kuvassa 10 esiintyvä @ManyToMany-annotaatiolla merkitty kenttä ei kuvaa itsessään mitään taulun kenttää vaan käänteistä suhdetta johonkin toiseen entiteettiin, jonka tietokantataulu sisältää vierasavaimen tähän tauluun. BaseEntity-luokka, josta Project-luokka perii, sisältää kaikille entiteeteille yhteisiä ominaisuuksia kuten primary key-muuttujan.

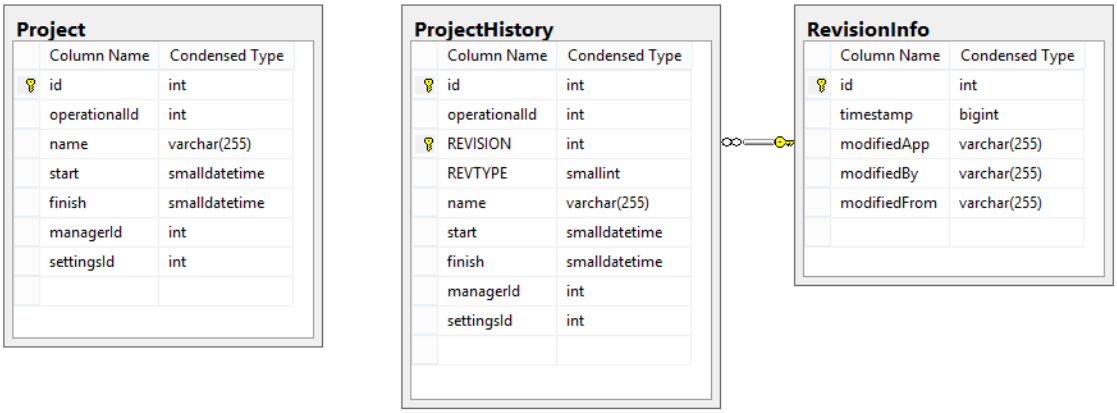
Joskus moni entiteetti voi sisältää samoja tietoja. Esimerkiksi voi olla normaaleja aikaraportteja, ylityöraportteja tai muita aikaraportteja. Silloin voi olla hyödyllistä tehdä abstrakti luokka, joka sisältää kaikkien raporttien yhteiset ominaisuudet kuten raportoijan ja raportoidun aikavälin. Sen lisäksi luodaan edelliselle luokalle aliluokkia, jotka sisältävät niille ominaiset piirteet. Esimerkkinä on kuvassa 11 esitelty abstrakti TimeReport-luokka, jonka aliluokkia ovat NormalTimeReport-, OvertimeReport- ja OtherTimeReport-luokat, joista kukin edustaa yhtä tietokantataulua.



Kuva 9. Luokkakaavio aikaraporttiluokkien periytymisestä

Tässä vaiheessa kaikki kolme raporttityyppiä ovat täysin samanlaiset, mutta ideana on, että myöhemmin jokaisella tulee olemaan toisistaan eroavia ominaisuuksia.

Tietokantataulujen auditointi on toteutettu Hibernaten Envers-lisäosaa käyttäen. Tämä tarkoittaa käytännössä, että jokaiseen auditoidun tietokantatauluun riviin tehtyjen operaatioiden tiedot tallennetaan erillisiin historiatauluihin (kuva 12).



Kuva 10. Project-tietokantataulu ja sitä vastaava historiataulu ja RevisionInfo-taulu

Historiataulut sisältävät kaikki samat kentät kuten niitä vastaavat auditoitavat taulut ja lisäksi tieto siitä, millainen operaatio tehtiin (insert, update vai delete) ja referenssi kaikille historiatauluille yhteiseen RevisionInfo-tauluun. Tämä taulu sisältää jokaisen entiteetin uuden version aikaleiman ja muita tietoja, jotka ovat vapaasti kustomoitavissa. Tälläisiä tietoja voisivat olla esimerkiksi muutoksen aiheuttaneen tekijän identiteetti tai IP-osoite, josta muutos tehtiin.

6.4 Tiedon validointi

Kaikki rajapinnan vastaanottamat resurssit viedään kaksitasoisen validointiprosessin läpi, jotta tietokantaan talletetun tiedon eheys ja bisnessääntöjen mukainen oikeellisuus voidaan varmistaa, koska ei voida luottaa, että rajapinnan vastaanottama tieto on täysin eheätä. Ensimmäinen validointiaskel ottaa kantaa vain rajapintaan lähetetyn tiedon rakenteeseen, ja se on toteutettu JSON-skeemavalidoinnilla. Jos skeemaa vasten validoidessa ei esiintynyt virheitä, siirrytään bisnesvalidointiin, jossa sääntömoottori evaluoi tiedon vasten joukon bisnessääntöjä. Jos virheitä ei ilmentynyt, voidaan uusi data tallettaa tietokantaan.

6.4.1 JSON-skeema

JSON-skeema on standardi, jonka tarkoituksena on kuvata JSON-olion rakennetta metatiedon avulla. Tiedon rakenteen oikeellisuus voidaan tarkistaa validoimalla sen skeemaa vasten. Skeema on myös hyödyllinen rajapintaa käyttävien ohjelmistojen ke-

hittäjille, koska skeeman avulla voidaan esimerkiksi luoda rajapintaa käyttävälle ohjelmalle käyttöliittymä, joka pystyy heti kertomaan käyttäjälle, jos jokin hänen antamansa tieto ei vastaa skeeman määrityksiä. Esimerkkinä on Person-entiteetistä serialisoitu JSON-olio ja sitä kuvaava skeema koodiesimerkissä 3.

```
{
    id: 1,
    operationalId: 12345,
    active: true,
    firstName: "Teppo",
    lastName: "Testeri",
    timeBalance: 125,
    email: "teppo.testeri@testaajat.fi"
}
{
    id: http://localhost:8080/time-app/api/v1/schemas/person.json,
    $schema: http://json-schema.org/draft-04/schema#,
    description: "schema for a Person entity",
    type: "object",
  - required: [
        "active",
        "firstName",
        "lastName",
        "email"
    ],
  - allOf: [
      - {
            $ref: "baseEntity.json"
        }
    ],
  - properties: {
      - active: {
            type: "boolean"
        },
      - firstName: {
            type: "string"
        },
      - lastName: {
            type: "string"
        },
      - timeBalance: {
            type: "integer"
        },
      - email: {
            type: "string"
        }
    }
}
```

Koodiesimerkki 3. Person-resurssin JSON-dokumentti ja sitä vastaava skeema

Skeeman id-muuttuja osoittaa paikan, missä kyseinen skeema sijaitsee. Dollarimerkillä merkitty schema-muuttuja kertoo taas, missä itse skeeman oma skeema eli meta-skeema sijaitsee. Sen osoitteesta näkee, että kyseessä on draft-versio, koska JSON-skeema on vielä keskeneräinen eikä siitä ole kehittynyt vakiintunutta standardia. Required-muuttujaan on listattu kaikki pakolliset kentät, jotka dokumentista pitää löytyä. AllOf-muuttuja kertoo, että objektista pitää löytyä kaikki sen sisällä määritetyt kentät. Tässä tapauksessa se sisältää viitteen toiseen skeematiedostoon, joka sisältää kaikille entiteeteille samat kentät kuten id:n. Viimeisenä skeemassa on listattuna kaikki Person-objektin muuttujat ja niiden tyypit.

6.4.2 Bisnessäännöt

Jos tieto validoitui oikein skeemaa vasten, validoidaan se vielä toisen kerran ennen tallettamista tietokantaan. Se tehdään käyttämällä bisnessääntöjä ja Drools-sääntömoottoria. Bisnessäännöt ovat joukko sääntöjä, jotka määrittävät tai rajoittavat, miten jokin organisaatio voi toimia [9]. Esimerkki bisnessäännöstä voisi olla, että tuntiraporttia kirjatessa raportin aloitusaika pitää olla ennen lopetusaikaa. Säännön voi esittää Droolsin ymmärtämässä muodossa koodiesimerkin 4 osoittamalla tavalla.

```
rule "Finish time is set before start time"
salience 110
    when
        tr : NormalTimeReport( start.isAfter(finish) )
    then
        errorList.add("Time report's finish time is before start time");
end
```

Koodiesimerkki 4. Bisnessääntö, joka nostattaa virheen, jos aikaraportin aloitusaika on lopetusajan jälkeen.

Säännöt toimivat hyvin johdonmukaisesti: kun when-osion sisältämä logiikka evaluoituu todenmukaiseksi, siirrytään suorittamaan then-osiossa olevaa koodia. Tässä tapauksessa se kirjaa virheviestin globaaliin listamuuttujaan. Valmiissa sovelluksessa tulee todennäköisesti olemaan tuhansia sääntöjä, jotka ohjaavat tiedon käsittelyä.

7 Loppusanat

Tämän insinöörityön tarkoituksena oli tutustua REST-arkkitehtuurimalliin ja toteuttaa sen periaatteita noudattava web-rajapinta PlanMill-toiminnanohjausjärjestelmälle. Myöhemmin työn aihe kuitenkin muuttui kattamaan koko web-sovelluksen arkkitehtuurin.

Lukiessani useita web-artikkeleita ja Fieldingin väitöskirjaa, totesin, että REST-mallista löytyy hyvin monta erilaista tulkintaa. Oli joitain kohtia, joissa yksi henkilö tulkitsee jonkin asian REST-mallia noudattavaksi, kun taas toinen väittää sen olevan täysin mallia vastaan. Koska Fielding ei ota väitöskirjassaan yhtään kantaa arkkitehtuurin toteutukselle, jättää se sille paljon tulkinnan varaa.

Itselläni ei ole kokemusta monesta web-arkkitehtuurimallista, joten en osaa verrata RESTiä mihinkään muuhun malliin kuin siihen, mitä lähdin tässä projektissa parantamaan. PlanMill Open API -rajapintaan verrattuna täysverinen REST-malli on kuitenkin huomattavasti helpommin lähestyttävä sen yhtenäisyyden ja selkeyden ansiosta. Julkisten REST-mallisten web-rajapintojen määrän viimeaikainen huima kasvu on varmasti suureksi osaksi tämän ansiota. Koska REST on hiljattain saanut paljon suosiota, on pinnalle noussut useita ohjelmia ja kirjastoja, joiden avulla rajapinnan suunnittelu ja toteuttaminen on hyvin helppoa. Tästä hyviä esimerkkejä ovat tässä projektissa käyttämäni Mulesoft Anypoint API Platform ja Oraclen Jersey-kirjasto.

Vaikka projektin aihepiiri laajeni huomattavasti työn aikana, ei projektin aikana esiintynyt mitään suurempia haasteita. Minulla oli entuudestaan hieman kokemusta REST-rajapinnan toteuttamisesta Oraclen Jersey-kirjastolla, joten alkuun pääseminen ei ollut hankalaa. Käyttämäni komponentit integroituivat sujuvasti yhteen ja niistä löytyi kattavaa dokumentaatiota ja vastauksia yleisiin ongelmiin internetistä. Lopputuloksena syntyi web-sovellus, joka kykenee hakemaan, luomaan, muokkaamaan ja poistamaan REST-arkkitehtuurimallin mukaisesti tuntiraportteja, validoimaan niiden sisältämän tiedon oikeellisuuden ja tallettamaan ne pysyvästi tietokantaan.

Insinöörityön ideana oli luoda proof of concept -mallinen toteutus, joka mielestäni onnistui hyvin, vaikka siitä jäi uupumaan yksi web-rajapinnoille oleellinen komponentti eli käyttäjien autentikaatio. Myös epäselväksi jäi, kuinka hyvin tämä arkkitehtuurimalli pystyy skaalautumaan kattamaan PlanMill-järjestelmän kokoisen sovelluksen.

Tämän insinöörityön alkuperäinen tavoite oli suunnitella ja toteuttaa PlanMill-toiminnanohjausjärjestelmälle uusi aidosti REST-mallinen web-rajapinta. Uusi rajapinta saatiin toteutettua, mutta sitä varten piti kehittää uusi sovellusarkkitehtuuri.

Uuden arkkitehtuurin suunnittelu sujui hyvin, koska pystyin soveltamaan siihen monia itselleni jo entuudestaan tuttuja komponentteja. Näistä komponenteista sain rakennettua pala palalta suuremman kokonaisuuden, josta lopulta muovautui täysvaltainen sovellusarkkitehtuuri.

Lopputuloksena syntyi täten proof-of-concept-mallinen kehys uudelle PlanMill-sovellusarkkitehtuurille, jonka painopisteenä on REST-arkkitehtuurimallia noudattava web-rajapinta. Mallitoteutus sisältää toiminnallisuudet tuntiraporttien kirjaamiseen, muokkaamiseen ja poistamiseen rajapintakutsujen avulla.

Insinöörityön aiheen radikaalisesta muutoksesta huolimatta työn toteuttaminen sujui hyvin eikä mitään suurempia ongelmia esiintynyt, joten mielestäni projektia voidaan pitää onnistuneena.

Lähteet

1 Fielding, Roy. Architectural Styles and the Design of Network-based Software Architectures 2000. Väitöskirja. <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>. Katsottu 31.3.2015.

2 Burke, Bill. RESTful Java with JAX-RS 2.0 2014. O'Reilly Media.

3 Apache Tomcat. Verkkosivu. Wikipedia. <http://en.wikipedia.org/wiki/Apache_Tomcat>. 13.3.2015. Katsottu 31.3.2015.

4 Java API for RESTful Web Services. Verkkosivu. Wikipedia. <http://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services>. 17.3.2015. Katsottu 31.3.2015.

5 Jackson Project Home @github. Verkkosivu. <https://github.com/FasterXML/jackson>. 4.3.2015. Katsottu 31.3.2015.

6 Hibernate Developer Guide. Verkkosivu. <http://docs.jboss.org/hibernate/orm/4.2/devguide/en-US/html/>. 28.1.2015. Katsottu 14.2.2015.

7 Hibernate (Java). Verkkosivu. Wikipedia. <http://en.wikipedia.org/wiki/Hibernate_(Java)>. 20.6.2014. Katsottu 3.7.2014.

8 Drools. Verkkosivu. Wikipedia. <http://en.wikipedia.org/wiki/Drools>. 27.6.2014. Katsottu 3.7.2014.

9 Business Rule. Verkkosivu. Wikipedia. <http://en.wikipedia.org/wiki/Business_rule>. 1.7.2014. Katsottu 3.7.2014.